Software Quality Journal manuscript No.
(will be inserted by the editor)

Coverage-Based Quality Metric of Mutation Operators for Test Suite Improvement

Pedro Delgado-Pérez · Louis M. Rose · Inmaculada Medina-Bulo

Received: date / Accepted: date

Abstract The choice of mutation operators is a fundamental aspect in mutation testing to guide the tester to an effective test suite. Designing a set of mutation operators is subject to a trade-off between effectiveness and computational cost: a larger mutation population might uncover more faults, but will take longer to analyse. With the aim of resolving this trade-off, several authors have defined an assortment of metrics to determine the most valuable operators. In this work, we extend an existing quality metric by incorporating an additional source of data, coverage information, and therefore investigate the extent to which mutants that often covered but rarely killed can improve the evaluation of mutation operators for the refinement of the test suite. As a case study, we analyse C++ class-level operators based on the new coverage-based quality metric to assess whether the original metric is enhanced. The results when selecting the best-valued operators show that this metric has great potential to help the tester in finding effective mutation operators. In comparison with the metric from which is derived, the use of coverage data allows to reduce the number of mutants but often losing fewer test cases and, in addition, retaining those that seem hard to design.

Keywords Software testing · mutation testing · quality metric · test suite improvement · test coverage

Pedro Delgado-Pérez
University of Cádiz. Escuela Superior de Ingeniería, Spain.
E-mail: pedro.delgado@uca.es
Tel.: +34 956 483243

Louis M. Rose
University of York. Department of Computer Science, UK.
E-mail: louis.rose@york.ac.uk

Inmaculada Medina-Bulo
University of Cádiz. Escuela Superior de Ingeniería, Spain.
E-mail: inmaculada.medina@uca.es

1 Introduction

Mutation testing (DeMillo et al., 1978) is regarded as a powerful mechanism to assess and improve the test suite effectiveness, but it is also known to involve a high cost that can preclude testers from making use of it. In this technique, the tester intentionally injects simple syntactic variations into the system under test (SUT) for the purpose of estimating the fault-revealing ability of a test suite. As a result, several versions of the original code are generated, called *mutants*, by means of *mutation operators* particularly defined for each programming language. When a test case executed against a mutant reveals its mutation (because of an observable difference in the output when compared to the one of the original program), we say that the test case is *effective* at finding faults. In this instance, the mutant has been *killed* (or the mutant is *dead*). On the contrary, the mutants that remain undetected or *alive* may provide the tester with the possibility of adding new test cases to the set.

Even in the case of small-sized programs, mutation testing is often an expensive process. Authors in this field have studied different methods to counteract the effect of the number of mutants generated in order to propel the technique for a definite uptake by practitioners (Jia & Harman, 2011; Offutt, 2011). The existence of mutants functionally *equivalent* to the original SUT also limits its applicability in practice; the tester will need to undertake the time-consuming task of determining which of the surviving mutants are equivalent and cannot be killed by any input. Under these circumstances, a well-designed set of mutation operators is a key factor for the success of the technique. Measuring the quality of mutation operators is one of the aims of researchers with the goal of identifying which are the most valuable. This information offers an opportunity towards the reduction of the often unacceptable cost of applying mutation testing. The definition of metrics giving us an approximation of how good mutation operators are is based on the assumption that some operators are more profitable than others. Studies in this regard range from simple metrics, like the *mutation score*, i.e., the ratio of dead mutants to non-equivalent mutants (Offutt et al., 1996), to more sophisticated ones addressing other desirable features, such as the cost that entails including each operator (Mresa & Bottaci, 1999) or the proportion of each kind of mutants that they produce (Smith & Williams, 2009; Estero-Botaro et al., 2010).

Recently, Estero-Botaro et al. (2015) formulated a promising metric with the aim of estimating the value of each mutation operator for a more efficient application of mutation testing in practice. To do that, this metric considers the number of test cases killing a mutant and the number of mutants killed by those test cases. Roughly speaking, this metric favours those mutation operators that generate mutants detected by few test cases, and penalises those operators that produce mutants killed by many test cases as well as those that usually produce equivalent mutants. Additionally, the metric attaches great value to operators with the potential to generate mutants that induce the design of test cases which might only be created by inspecting few other

mutants. Thus, the study of the empirical results based on this metric can serve to foresee which operators will be more likely to help the tester mainly in designing *high-quality test cases*: those test cases that detect non-trivial faults and are hard to create. That means that, instead of measuring the ability of the test suite to detect faults, this quality metric has great potential in targeting the specificity of the test cases for the improvement of the test suite, as shown in a recent study applying selective mutation based on this metric (Delgado-Pérez et al., 2017b). Concretely, the authors highlight that 40% of the set of mutants can be saved with a loss in the percentage of test cases under 6% on average. Many of those mutants saved are equivalent, thereby reducing not only the cost of mutant execution, but also the cost of mutant analysis.

While the *coverage analysis* of the test suite has traditionally been used to avoid unnecessary executions in those mutants not covered by some of the test cases (Schuler & Zeller, 2009; Papadakis & Malevris, 2011; Derezińska & Szustek, 2012) (i.e., when the test cases do not execute the mutated statements), Inozemtseva & Holmes (2014) took advantage of this information in a recent study assessing the correlation between coverage and test suite effectiveness. They provided a more restricted version of the mutation score by discarding from the calculation those mutants executed by none of the test cases in the test suite. This measurement is useful to compare test suites with different code coverage since it does not consider that a test suite is unable to reveal a mutation that is not even executed. As such, the metric captures how thoroughly the test suite exercises the code that it actually covers.

Problem: The quality metric by Estero-Botaro et al. (2015) overlooks that not all mutants are exercised by all test cases, as raised by Inozemtseva & Holmes (2014). In fact, only the test cases that actually reach a mutated element might be able to reveal the defect injected into the SUT. A comparison of the number of test cases killing a mutant with the number of test cases reaching it could lead to a more robust metric that helps design better test suites at a lower cost.

Contribution: In this paper, we present a novel metric for measuring the quality of mutation operators that incorporates *coverage data* (i.e., which mutants are reached by which test cases) into the calculations. This metric is derived from the notion of quality by Estero-Botaro et al. (2015) and, therefore, it can be used as a method to know the best operators for the refinement of a test suite with high-quality test cases. Intuitively, mutants killed by few of the test cases that reach them should be more valuable than those mutants killed by many of the test cases that execute them.

As a means to evaluate the proposed metric, we make use of *selective mutation* (Wong & Mathur, 1993), a well-known cost reduction technique which has been applied following different approaches (Offutt et al., 1993, 1996). With selective mutation, the cost is eased by trying to exclude some of the operators in the set without significant loss of effectiveness. However, we do not follow the traditional approach of selecting a representative subset of the whole set

of operators, but one tailored to the purpose of this quality metric, which we call *test-quality selective mutation*. The aim when applying test-quality selective mutation is to give preference to mutation operators of high quality so that the test suite is efficiently improved mainly with high-quality test cases. Consequently, we do not calculate the mutation score, but new measurements (test suite *size* and *specificity*) in order to estimate the quality of the test suite that this selective approach could help us form. As a continuation of our research line applying mutation testing to C++ (Delgado-Pérez et al., 2015, 2017b), we conduct our experiments on class-level operators for this language.

The main contributions of this paper are:

- **A novel quality metric to measure operator effectiveness in the refinement of a test suite that takes into account the coverage of the test suite**. We call the metric including this information *coverage-based metric*. This quality metric allows for a more precise evaluation of the quality of mutation operators than the original quality metric (from which our metric is derived). We have also defined new terms to classify mutants regarding the coverage information, such as *coverage-based resistant* mutant or *difficult to reach* mutant.
- **An evaluation of the coverage-based metric**, using as a case study class mutation operators in C++ and test-quality selective mutation. We perform test-quality selective mutation after ranking the operators based on the metric. By removing some of the operators which are at the bottom of the classification sorted by the quality metrics, the results show that we assume a low loss in the percentage of test cases in general.
- **A comparison between the coverage-based quality metric and the original quality metric**. We define two new measurements, test suite size and specificity, to estimate the effectiveness when applying test-quality selective mutation. The comparison between metrics by using these measurements reveals that the coverage-based metric allows maintaining more test cases than the original metric overall when some of the operators are discarded. In addition, the new metric exhibits even more ability to retain specific test cases than the original one.

Scope: It is the goal of this paper:

- To define the coverage-based quality metric and to present how it operates.
- To evaluate whether the coverage-based metric outperforms the original metric thanks to the addition of coverage information.

The remainder of this paper is structured as follows. While the next section provides the motivation for the use of quality metrics and explanations on how the original quality metric works, Section 3 reasons how the coverage information can improve that metric and why a new selective strategy and new measures are required. Section 4 is devoted to the coverage-based quality metric and the new concepts related to coverage information. Section 5 shows the setup of the conducted experiments, and Section 6 analyses in depth the empirical results. Section 7, Section 8 and Section 9 address threats to validity,

final remarks and related work respectively. Finally, the main findings and possible research lines for the future are shown in the last section.

2 Quality Metrics

2.1 Motivation for the Use of Quality Metrics

Mutation testing: Because of the high cost that applying mutation testing entails (mutant execution and analysis), the reduction of the expenses has become a complementary goal when using this technique. In this context, the definition of quality metrics to estimate the utility of mutation operators can play an essential role: they can be used as a basis for restricting or excluding the least effective operators and consequently reducing the testing effort. In a recent work, Delgado-Pérez et al. (2017b) showed that considering a quality metric can be highly valuable for the application of selective mutation. In general, the experiments reported significant savings in the number of mutants (including equivalent ones) with a low loss in the number of test cases with respect to a test suite that kills all non-equivalent mutants. The study by Delgado-Pérez et al. (2017b) also supports the fact that any mutation operator can generate valuable mutants, as raised in previous works (Kurtz et al., 2016). However, their experiments also show evidence that giving preference to mutation operators over others (based on a ranking experimentally determined) is a better choice than selecting mutants from all operators with equal probability. This fact supports the application of a cost reduction technique following quality metrics.
Coverage analysis: Regarding the coverage-aware metric presented in this paper, it is important to note that several mutation tools already analyse the test suite coverage to reduce the cost (Schuler & Zeller, 2009; Papadakis & Malevris, 2011), as it will be commented later on in Section 3.2. As a result, these tools do not generate or execute a mutant (or a mutant against particular test cases) when the mutation is not covered by the test suite in practice. However, this tool improvement does not serve the same purpose as the coverage-quality metric proposed in this paper. Indeed, the coverage-based metric is used in advance in order to know the best operators for test suite improvement. The information derived from that previous analysis is then used in practice to produce a lower number of mutants. As such, while taking into account the test suite coverage in practice can reduce the need for execution (and barely the need for mutant analysis), the quality metric has the potential to greatly reduce both mutant execution and mutant analysis.

2.2 Original Quality Metric

Recently, Estero-Botaro et al. (2015) provided a definition of quality for mutants and operators with the goal of searching for specific test cases. Their

quality metric punishes the existence of equivalent mutants as well as takes into account a twofold aspect: the number of test cases that kill a mutant and the number of mutants that those concrete test cases kill in turn. Therefore, this metric considers that a mutant will assist a tester in designing *high-quality test cases* not only when there are few test cases killing it, but also when there are few mutants killed by those test cases. In other words, the fewer the mutants that are able to guide the tester in creating a test case, the more specific and hard to design is that test case. The metric by Estero-Botaro et al. (2015) allows estimating the effectiveness of mutation operators and, as such, we base the work that will be presented in the next sections on this metric. We will refer to this metric as *original quality metric*.

Nevertheless, other metrics have been proposed following different approaches. The first evaluations of operator effectiveness aimed at obtaining sufficient sets of mutation operators (i.e., subsets of operators that accurately predict the adequacy of the whole set of operators) by removing both the most prolific operators (Offutt et al., 1993) and those of the same category (Offutt et al., 1996) (based on the syntactic elements that they mutate). If the mutation score in the reduced set of operators is the same as in the original set after the selective strategy, the effectiveness of the technique remains high and those operators are not actually necessary.

Mresa & Bottaci (1999), in addition to calculating the mutation score of Fortran operators, also used the cost of applying them (test data generation and equivalent mutant identification) for a more accurate measurement of the trade-off of including each operator. Estero-Botaro et al. (2010) analysed a set of WS-BPEL operators with a focus on the quantitative dimension: number of invalid, equivalent, easy to kill and resistant mutants generated. In their study, they made use of the notion of quality that Derezińska (2006) took to assess C# class operators, which computes how effective are test cases in killing mutants. Smith & Williams (2009) went a step beyond when classifying mutants; they categorised the mutants depending on the outcome during successive attempts to kill them: the mutants killed by test cases specifically designed to detect them are more interesting than those killed by the initial test suite. As for object orientation, Hu et al. (2011) also proposed a metric, called *mutation operator strength*, to estimate the quality of Java class operators as the minimal number of tests needed to kill the set of non-equivalent mutants.

2.3 Execution Matrix

In order to calculate the quality metrics detailed in the next sections, each mutant needs to be run against each of the test cases in the test suite to produce a result. That result depends on the behaviour of the mutant when compared with the original program. To that end, the mutation tool produces as an output what Estero-Botaro et al. (2015) call *execution matrix*. As can be seen in Figure 1, the rows in the execution matrix represent the mutants and the columns represent the test cases. Thus, a value $v_{x,y}$ is the result of

the execution of the mutant x against the test case y. We identify a mutant that is killed by a test case with the value 1, and 0 otherwise. We will show an example of execution matrix to illustrate the quality metrics later on, as we require its information to perform the calculations.

	test_1	test_2	...	test_m
mutant_1	$v_{1,1}$	$v_{1,2}$	...	$v_{1,m}$
mutant_2	$v_{2,1}$	$v_{2,2}$	...	$v_{2,m}$
...	...	...	...	...
mutant_n	$v_{n,1}$	$v_{n,2}$	...	$v_{n,m}$

Fig. 1 Execution matrix format

2.4 Nomenclature

The symbols used in the following sections to refer to terms related to mutation testing are as follows:

- M - set of valid mutants. As mentioned in the previous subsection, we need to run each mutant and form an execution matrix to compute the quality metrics. Since invalid mutants (i.e., mutants that violate the language rules) cannot be executed, they are not included in M.
- E - set of equivalent mutants.
- D - set of dead mutants.
- T - an *adequate* and *minimal test suite* for the set of mutants D. A test suite is adequate when every non-equivalent mutant is killed by some test cases, and non-redundant when every test case is necessary to keep the adequacy. Additionally, an adequate and non-redundant test suite is labelled as minimal when it contains the minimum number of test cases keeping the adequacy for the full set of mutants (the definition of minimum test suite provided by Ammann et al. (2014) represents the same concept). If T is an adequate test suite for D, it is also adequate for M.
- K_m - set of test cases killing the mutant m.
- C_t - set of mutants killed by the test case t.

Similarly, we will use analogous notation regarding an operator:

- M_o - set of valid mutants produced by the mutation operator o.
- D_o - set of dead mutants within M_o.
- T_o - an adequate and minimal test suite for the set of mutants D_o (again, T_o is therefore adequate for M_o).

2.5 Definition of the Original Quality Metrics

Estero-Botaro et al. (2015) defined the notion of *quality of a mutant* m as follows:

$$Q_m(M,T) = \begin{cases} 0, & m \in E \\ 1 - \dfrac{1}{(|M| - |E|) \cdot |T|} \displaystyle\sum_{t \in K_m} |C_t|, & m \in D \end{cases} \quad (1)$$

By way of explanation of the metric, an equivalent mutant ($m \in E$) will be punished with the minimum value (0), whilst each dead mutant ($m \in D$) will be assigned a different value depending on how difficult is to produce test cases to kill it (the higher the value, the better is that mutant).

The metric Q_m provides the foundations for two further metrics regarding an operator:

- **Quality of a mutation operator o:**

$$Q_o(M_o, T_o) = \frac{1}{|M_o|} \sum_{m \in M_o} Q_m(M_o, T_o) \quad (2)$$

  Therefore, the quality of a mutation operator is based on the quality of their mutants.
- **Quality of the dead mutants produced by a mutation operator o:**

$$Q_{D_o}(D_o, T_o) = \frac{1}{|D_o|} \sum_{m \in D_o} Q_m(D_o, T_o) \quad (3)$$

Notice that the metric Q_m is dependent on the test suite (T) and the set of mutants (M). The authors of this metric consider that T should be adequate and minimal for the reliability of the results (see Section 2.4). As can be seen from Equations 2 and 3, the metrics Q_o and Q_{D_o} are computed using the set of mutants from that operator (M_o and D_o respectively). As such, these two quality metrics regarding a mutation operator are completely independent of the mutants generated by other operators. Thus, just considering the mutants from that operator for the calculations ensures that the operator will always show the same behaviour (with the same test suite) regardless of the rest of operators.

As a simple example, consider the execution matrix (see Section 2.3) that appears in Figure 2[1]. This matrix is a fragment with the four mutants generated by one operator and an adequate and minimal test suite for those mutants (formed by three test cases). Being:

- $|M| = 4$, $|E| = 0$, $|T| = 3$.
- $K_{m_1} = \{test_1\}$, $K_{m_2} = \{test_2, test_3\}$, $K_{m_3} = \{test_3\}$ and $K_{m_4} = \{test_2\}$.
- $C_{test_1} = \{m_1\}$, $C_{test_2} = \{m_2, m_4\}$ and $C_{test_3} = \{m_2, m_3\}$.

[1] From now on, we will refer to mutant x as m_x for the sake of simplicity.

Mutant	$test_1$	$test_2$	$test_3$
m_1	1	0	0
m_2	0	1	1
m_3	0	0	1
m_4	0	1	0

Fig. 2 Execution matrix to illustrate the *original quality metric*.

then, the quality of these mutants is:

- $Q_{m_1} = 1 - 1/((4-0)\cdot 3) = 0.92$
- $Q_{m_2} = 1 - 4/((4-0)\cdot 3) = 0.67$
- $Q_{m_3} = 1 - 2/((4-0)\cdot 3) = 0.83$
- $Q_{m_4} = 1 - 2/((4-0)\cdot 3) = 0.83$

The mutant m_1 is the most valued because it is only killed by one test case ($test_1$), which kills no other mutants. Contrarily, m_2 is of lower quality since it is killed by two test cases ($test_2$ and $test_3$), which in turn kill another mutant (m_4 and m_3 respectively). m_3 and m_4 are valued the same (0.83) as they both are killed by a single test case that kills another mutant.

Finally, knowing these values, we can measure the quality of the mutation operator that produced those mutants as follows:

$$Q_o = (0.92 + 0.67 + 0.83 + 0.83)/4 = 0.81$$

As a final remark, $Q_{D_o} = Q_o = 0.81$ because there are no equivalent mutants ($M_o = D_o$).

2.6 Mutant Categorisation

Also with regard to the quality of mutants, Estero-Botaro et al. (2010) defined different terms to categorise mutants which make use of the information in the execution matrix:

- **Trivial mutants** are killed by every test case in the test suite. They can be identified as a row filled with the value 1. Note that the original term for this kind of mutants is *weak mutant*, but we use trivial instead (following Just & Schweiggert (2015)) to distinguish trivial mutants from *weak mutation* (Papadakis & Malevris, 2011).
- **Resistant mutants** are killed by a single test case. They can be identified as a row filled with the value 0 except for one entry with the value 1. The set of resistant mutants is denoted by R.
- **Resistant hard to kill mutants** are killed by a single test case that kills no other mutants. They can be identified as a row filled with the value 0 except for one entry y with the value 1. At the same time, the rest of the entries in the column y are filled with the value 0. The set of resistant hard to kill mutants is denoted by H.

While trivial mutants model faults that are easy to detect, resistant and resistant hard to kill mutants simulate non-trivial situations. Thus, trivial and resistant mutants represent respectively the worst and the best-valued mutants by the original quality metric. The rest of the mutants lie within that range and receive a mark according to the number of test cases and mutants killed by those test cases. The mutants m_3 and m_4 (see Figure 2) are resistant mutants, and the mutant m_1 is a resistant hard to kill mutant. Note that we cannot find a trivial and a resistant hard to kill mutant simultaneously in the same execution matrix.

3 New Approach to Estimate the Quality of Mutation Operators

3.1 Motivation for the Improvement of the Original Quality Metric

As mentioned before, the original quality metric Q_m is dependent on the test suite used (T), which should be adequate and minimal. This metric is also dependent on the number of mutants generated (M). Thus, the same size of the test suite ($|T|$) and the set of mutants ($|M|$) is used to calculate Q_m for every mutant (see Equation 1).

However, this can lead to a deviation in the results which is related with the next statement made by their authors (Estero-Botaro et al., 2015): "Please notice that a resistant mutant can also be trivial in the degenerate case that $|T| = 1$". In the same sense, a mutation will be reached by some test cases while it will not be exercised by others. Thus, being the number of test cases greater than one, in the most extreme case a resistant mutant might only be reached by the test case that kills it; in that instance, the size of the test suite with respect to that mutant is actually one. This is even more relevant when testing object-oriented systems, where test cases are designed as scenarios, each one of them exercising different methods belonging to different classes. Thus, a test scenario usually addresses a class or subset of classes, while it does not cover the rest of classes.

According to this, **a mutant executed by many test cases but killed by few of them is regarded as more difficult to kill (and should be more valued) than other mutants killed by most of the test cases covering the mutation**. That is, knowing which mutants are often covered but rarely killed may result in valuable information for the evaluation of mutation operators. Consequently, the fact that not all the test cases reach all the mutants leads us towards a new approach for a finer-grained measurement of the quality of a mutant. In order to estimate the quality of a particular mutant, a quality metric using the coverage information of the test suite should only consider the set of test cases reaching the mutant as well as the set of mutants reached by those test cases at the same time (instead of the size of the whole test suite and the size of the whole set of mutants, as the original quality metric does).

3.2 Use of Coverage Information

To optimise the mutant generation or execution stages, some mutation tools harness the coverage information of the test suite. Coverage information is used to avoid the injection of mutations when they are not executed by any of the test cases. Likewise, avoiding the execution of test cases against several mutants is also possible by analysing the coverage information. At this point, it is important to note that this tool improvement is not incompatible with the use of the quality metric with coverage information. The coverage-based metric is used in advance, increasing the knowledge about the most valuable operators for test suite improvement. The generation of mutants from those best-valued operators can be later favoured in the mutation tool. Finally, the mutation tool in practice could avoid the generation/execution of uncovered mutants from the subset of mutants generated to further reduce the cost.

CREAM for the C# programming language (Derezińska & Szustek, 2012) integrates the *IBM Rational PureCoverage* tool to obtain coverage data and be able to automatically detect mutations covered by none of the test cases, thus avoiding the need to inspect them when classifying mutants as equivalent or not. *Javalanche* (Schuler & Zeller, 2009) collects which statements of the Java code are covered by each test case, thereby only executing the test cases that reach each mutant. *PROTEUM/IM* (Vincenzi et al., 2006) instruments the original code by using a descriptor language that assists in storing the nodes of the program graph reached by each test case, thus speeding up the mutant execution later. *MutPy* for Python (Derezinska & Halas, 2014) also implements a coverage analysis to reduce mutants generated and test cases executed. *Bacterio* (Mateo & Usaola, 2012) uses the mutant schemata technique (Untch et al., 1993) in order to reduce compilation times. This tool includes another technique, called *MUSIC*, that reduces mutant execution by adding extra code to the mutant schemata, and that avoids infinite loops at the same time.

Unlike those tools, which only analyse the coverage of the statements, Papadakis & Malevris (2011) used dynamic symbolic evaluations to implement weak mutation and obtain a more accurate mechanism to reduce test case executions. To this end, a control flow graph containing the full set of mutants and the infection conditions is generated. The execution of test cases can be later filtered not only when they do not cover a mutant, but also when it is known that the system is not infected.

Inozemtseva & Holmes (2014) recently leveraged the coverage information in a very different way from the above mentioned works. They devised a metric called *normalized effectiveness measurement*, which is similar to the mutation adequacy score, but using the number of non-equivalent mutants covered by the test suite instead of the whole set of non-equivalent mutants. The metric helped them to produce evidence that the test suite effectiveness is not strongly correlated with coverage criteria. We will make use of this approach in our work to strengthen the original quality metric explained in the previous section.

3.3 Test-Quality Selective Mutation

Selective mutation is a technique based on the omission of some of the operators in the set to reduce the cost while retaining effectiveness. The goal in selective mutation is to find a reduced set of operators which is representative of the full set of operators. That representativeness is calculated through the test suite obtained by means of the subset of operators selected; we obtain a sufficient set of operators if the mutation score of that test suite when measured against the original set of operators correlates with the mutation score associated with the reduced set of operators. This is the approach to selective mutation when it comes to evaluating the fault detection capability of the test suite. However, as mentioned before, the presented quality metric focuses on test suite improvement with high-quality test cases. Therefore, it is not the purpose of the quality metric to value operators for their potential to predict the mutation score of the full set of operators. This was pointed out by Delgado-Pérez et al. (2017b), who proposed to assess operators in a different way depending on whether the test suite is being evaluated or refined.

As a consequence, we should not apply selective mutation and compute the mutation score (as traditionally done) to evaluate the effectiveness of the quality metric. We can illustrate with a simple example why a new approach related to test quality is required. Consider again the executing matrix in Figure 2. If we had to select only one mutant based on the quality metric, we would select the mutant m_1 because it is a resistant hard to kill mutant (see Section 2.5). Therefore, $test_1$ does not kill any other mutants, so the mutation score is low but we are retaining a test case which is not easy to design. As it can be seen, the mutation score is not an appropriate method to measure the effectiveness when using this quality metric. In our approach, whether the test suite kills a large number of mutants is unimportant as we just seek a set of operators which is effective in the refinement of the test suite with specific test cases.

Instead of traditional selective mutation, we propose a *test-quality selective mutation* with appropriate metrics to measure the quality of the test suite. By applying test-quality selective mutation we do not seek for a representative subset of operators, but for a subset that allows us to leverage the information of surviving mutants in such a way that the test suite is enhanced with as many high-quality test cases as possible. We define the following metrics to compute its effectiveness:

1. **Test suite size:** *Percentage of test cases loss* when compared to the original adequate and minimal test suite. Being n a number to represent the n best-valued operators selected by the quality metric (represented by $o_1 ... o_n$), it is measured as follows:

$$\frac{|T| - |T_{o_1 ... o_n}|}{|T|} \times 100$$

The lower the percentage, the fewer test cases we are losing because of removing the rest of operators which are not in $\{o_1...o_n\}$. Going back to the same example, $test_1$ is the only test case (out of three test cases) that the selected mutant m_1 would be able to induce. Therefore, the percentage of test cases loss with respect to the adequate and minimal test suite would be $(3 - 1/3) \times 100 = 66.6\%$. If we had selected m_4, the loss would be the same, as this mutant can also lead to the design of just one test case ($test_2$).

2. **Test suite specificity:** *Average percentage of mutants killed* by the test cases in the test suite. Again, being $o_1...o_n$ the n best-valued operators selected by the quality metric, it is measured as follows:

$$\frac{\sum_{t \in T_{o_1 \ldots o_n}} (|C_t| / |M|) \times 100}{|T_{o_1 \ldots o_n}|}$$

The lower the percentage, the more specific are the test cases in the test suite formed with the operators $o_1...o_n$, as those test cases may only be generated by inspecting few of the mutants in the full set. In our example, the test suite size was the same selecting either m_1 or m_4. However, $test_1$ is a more specific test case than $test_2$. This is reflected by the test suite specificity: $test_1$ and $test_2$ kill $((1/4) \times 100/1) = 25\%$ and $((2/4) \times 100/1) = 50\%$ of the mutants respectively.

In general, the best situation is to retain all those mutants that can lead to the design of a mutant-adequate test suite (therefore, the test suite loss is 0%). Otherwise, the loss of some test cases is not so problematic if that affects the specificity positively (i.e., the average percentage of mutants killed by the test cases decreases); that means that the generated mutants can still lead to the design of the high-quality test cases, which is the goal of using the quality metric in this paper.

Test-quality selective mutation removing complete mutation operators from the execution (operator-based selection) is used in the experiments in this paper in Section 5. Recent studies suggest that mutant-based selection can be a better choice than operator-based selection (Zhang et al., 2010; Gopinath et al., 2015). On the basis of this idea, a rank-based strategy was recently proposed (Delgado-Pérez et al., 2017b), in which the probability of selecting mutants from an operator is proportional to its position in the operator ranking (formed with the values that the quality metric gives to each operator). However, we apply operator-based selection in these experiments to avoid the stochastic factor of the rank-based strategy, which should help to observe more precisely the actual difference between the original and the proposed coverage-based metric.

4 Coverage-Based Quality Metric of a Mutation Operator

4.1 Previous Definitions

Before presenting the new quality metric, we need to clarify the meaning of coverage of the test cases with respect to the mutants. For this purpose, we define several concepts connecting mutants and coverage of the test suite as follows:

Definition 1 (Reaching a mutant) Let *reaches* be the binary relation such that t reaches mutant m iff the mutated statement in m is executed by test case t.

Note that, for a mutant m to be killed by test case t, t must reach m.

Definition 2 (Mutants reached by test case t) Let M_t be the set of mutants reached by test case t:

$$M_t = \{m \in M \mid t \text{ reaches } m\} \tag{4}$$

Therefore, $M_t \subseteq M$.

Definition 3 (Equivalent mutants reached by test case t) Let E_t be the set of equivalent mutants reached by test case t:

$$E_t = \{m \in E \mid t \text{ reaches } m\} \tag{5}$$

Therefore, $E_t \subseteq E$.

Definition 4 (Non-equivalent mutants reached by test case t) Let N_t be the set of non-equivalent mutants reached by test case t:

$$N_t = M_t \setminus E_t \tag{6}$$

Therefore, $N_t \subseteq M \setminus E$.

Definition 5 (Test cases reaching mutant m) Let T_m be the set of test cases reaching mutant m:

$$T_m = \{t \in T \mid t \text{ reaches } m\} \tag{7}$$

Therefore, $T_m \subseteq T$.

Definition 6 (Coverage matrix) A coverage matrix of size $|M| \times |T|$ describes which mutants are reached by which test cases.

A coverage matrix uses the same format as the execution matrix shown in Figure 1. In this case, a value $v_{x,y}$ reflects whether the mutant x was reached by the test case y; a symbol 'x' will be marked in the coverage matrix to register which test cases reach each mutant.

4.2 Example of Execution Matrix and Coverage Matrix

Consider the execution matrix EM (see Figure 3) with the results of running a set of eight mutants against a test suite with five test cases, which is adequate and minimal (therefore, m_8 is an equivalent mutant). As can be seen, m_4 and m_7 are resistant mutants since they are only killed by the test case $test_5$ and $test_1$ respectively. In both mutants, the test case killing them also kills another mutant ($test_1$ kills m_1 and $test_5$ kills m_2), so the original metric will give both mutants the same value (see the same case in the example in Section 2.5 with m_3 and m_4).

Now consider a coverage matrix CM (see Figure 4) of the same size than EM (8×5). In the matrix CM, the test cases reaching each mutant are marked with 'x', as explained in the definition of coverage matrix (see Definition 6). This matrix shows that m_4 might only be killed by two test cases ($test_1$ and $test_5$), while m_7 might be killed by four test cases in the test suite ($test_1$, $test_2$, $test_3$ and $test_5$). Therefore, m_7 seems to be more valuable than m_4 because it appears to be harder to kill.

Mutant	$test_1$	$test_2$	$test_3$	$test_4$	$test_5$
m_1	1	0	1	0	0
m_2	0	0	1	0	1
m_3	0	0	1	0	0
m_4	0	0	0	0	1
m_5	0	0	0	1	0
m_6	0	1	0	0	0
m_7	1	0	0	0	0
m_8	0	0	0	0	0

Fig. 3 Execution matrix EM.

Mutant	$test_1$	$test_2$	$test_3$	$test_4$	$test_5$
m_1	x	-	x	x	-
m_2	-	-	x	-	x
m_3	-	x	x	-	-
m_4	x	-	-	-	x
m_5	-	-	-	x	-
m_6	x	x	x	x	x
m_7	x	x	x	-	x
m_8	-	-	x	-	x

Fig. 4 Coverage matrix CM.

An analogous approach can be taken regarding the mutants reached by each test case for a finer measurement. Following the double perspective of the original metric (see Section 2.2), we are also interested in finding mutants

which induce the creation of test cases that might only be designed by reviewing few mutants. Therefore, the fact that a test case kills only few of the mutants reached by that test case is a valuable information for the new quality metric. As an example, $test_4$ reaches three mutants (m_1, m_5 and m_6), but it is only able to kill m_5.

4.3 Coverage-Based Mutant Categorisation

In this section, we will redefine the terms *resistant* and *resistant hard to kill mutant* provided by Estero-Botaro et al. (see Section 2.6) according to the particularities of the new metric considering the coverage of the test suite. The definition of these concepts is important as they delimit the values that the coverage-aware metric can assign to mutants. It should be clarified that we do not define an analogous concept "coverage-based trivial mutant" because it would be equivalent to "trivial mutant".

Definition 7 (Coverage-based resistant mutant) A mutant m is resistant considering the coverage of the test suite when the mutant is reached by all the test cases in the test suite, but only one test case kills it. Let RC be the set of coverage-based resistant mutants:

$$RC = \{m \in M \mid |K_m| = 1 \wedge T_m = T\} \tag{8}$$

Therefore, it is clear that $RC \subseteq R$ (recall that R is the set of resistant mutants, as mentioned in Section 2.6). A mutant belonging to RC is more difficult to kill than a mutant belonging to $R \setminus RC$.

Definition 8 (Coverage-based resistant hard to kill mutant) A mutant m is resistant hard to kill considering the coverage of the test suite when the mutant is reached by all the test cases in the test suite, and it is killed by only one test case that reaches the rest of the mutants but kills none of them. Let HC be the set of coverage-based resistant hard to kill mutants:

$$HC = \{m \in RC \mid \forall t \in K_m \, |C_t| = 1 \wedge N_t = M \setminus E\} \tag{9}$$

Therefore, it is clear that $HC \subseteq H$ (recall that H is the set of resistant hard to kill mutants). A mutant belonging to HC is more difficult to kill than a mutant belonging to $H \setminus HC$.

It is important to note that the provided definitions only serve as reference points and the quality metric will assign a value to each mutant according to its corresponding part in the execution and coverage matrices. Therefore, even though a mutant killed by one test case and reached by all except one of the test cases does not exactly match the definition of coverage-resistant mutant, it will obtain a lower but quite similar value.

We can also define a new type of mutant, *difficult to reach mutant*, which will play an important role in the new metric. Going back to the matrix CM in the previous section, since m_5 is only reached by one test case ($test_4$), we may deem that this mutant is not easy to cover. These mutants that are difficult to reach normally affect fragments of code barely tested or used within the code of the program. Note that, while coverage-based resistant mutants are easy to reach but difficult to kill, we do not know whether difficult to reach mutants are easy or difficult to kill. In other words, we have no certainty about whether or not new test cases reaching a difficult to reach mutant would be able to kill it. Because of the lack of coverage information, we should not use the coverage of these mutants to value their quality. Thus, we define the term of difficult to reach mutant in order to include a special case for this kind of mutants in the definition of the coverage-based quality metric of mutation operators, as we will see later on in this paper.

As a remark before providing the definition of difficult to reach mutants, we might consider the number of test cases reaching a mutant ($|T_m|$) as the only coverage information that is meaningful to determine whether a mutant is difficult to reach or not. However, for the sake of consistency with the twofold approach of the metric, the number of non-equivalent mutants reached by each test case ($|N_t|$) is also regarded as part of the coverage information related to a mutant. Returning to the example, we not only consider that m_5 is reached by $test_4$, but also that this test case reaches m_1, m_5 and m_6.

Definition 9 (Difficult to reach mutant) Let MCOV be a constant representing the minimum number of coverage data required so that the coverage is regarded as significant. A dead mutant m is difficult to reach when the sum of the test cases reaching m and the non-equivalent mutants reached by those test cases is less or equal than MCOV. Let DR be the set of difficult to reach mutants:

$$DR = \{m \in D \mid \sum_{t \in T_m} |N_t| \leq MCOV\} \tag{10}$$

Note that we limit this definition to dead mutants because whether equivalent mutants are difficult to reach or not is uninteresting. Finally, we should note that, as $T_m \subseteq T$, in the degenerate case that $\sum_{t \in T} |N_t| \leq MCOV$, all the mutants in D will also be classified as difficult to reach.

4.4 Coverage-Based Quality Metric

Bearing in mind the definitions and explanations about coverage in the previous sections, we can now provide a definition for a new quality metric considering the coverage of the test suite:

Definition 10 (Coverage-based quality metric of a mutant)

$$QC_m(M,T) = \begin{cases} 0, & m \in E \\ 1 - \dfrac{1}{\sum\limits_{t \in T_m} |N_t|} \sum\limits_{t \in K_m} |C_t|\,, & m \in D \setminus DR \\ 1 - \dfrac{1}{\sum\limits_{t \in T} |N_t|} \sum\limits_{t \in K_m} |C_t|\,, & m \in DR \end{cases} \tag{11}$$

Note that the only difference between $m \in D \setminus DR$ and $m \in DR$ is that the former case uses T_m and the latter uses T. That is, difficult to reach mutants are valued with respect to the test suite without considering the coverage, in line with the explanations previously provided.

This formal definition of coverage-based quality of a mutant contemplates three different cases:

- Case $m \in E$ (equivalent mutants):
Equivalent mutants are equally penalised with a 0.

- Case $m \in D \setminus DR$ (dead mutants not considered as difficult to reach):
This metric meets the following conditions:

1. $K_m \subseteq T_m$ and $C_t \subseteq N_t$. This means that, whenever there is a value 1 in the execution matrix, there should be a value 'x' in the coverage matrix.
2. Subsequent to the preceding point, $\sum_{t \in K_m} |C_t| \,/\, \sum_{t \in T_m} |N_t| \leq 1$. This implies that $QC_m \geq 0$, exactly as the original quality metric of a mutant (Q_m).
3. Let $T_m = T$, that is, when the row corresponding to mutant m is filled with 'x' in the coverage matrix. Also let $|N_t| = |M| - |E|$, that is, when the mutants in $M \setminus E$ are marked with 'x' in the column corresponding to test case t in the coverage matrix. Then, $QC_m = Q_m$.

This last point shows the conditions under which the original and the coverage-based quality metric assign the same value to a mutant $m \in D \setminus DR$.

- Case $m \in DR$ (difficult to reach mutants):
In the case of difficult to reach mutants, even when we do not know whether a difficult to reach mutant is difficult to kill as well (see Section 4.3), we seek to maintain this mutant as it might provide a necessary test case. Indeed, when $|T_m| = 1$, a difficult to reach mutant is considered to be resistant by the original definition by Estero-Botaro et al. (see resistant mutant in Section 2.6); resistant mutants receive a high mark by the original metric. Therefore, the coverage-based metric has been devised to assign the same value to difficult to reach mutants and coverage-based resistant mutants as they are equally useful to refine a test suite: $T_m = T$ in the latter kind of mutant, so the cases $m \in D \setminus DR$ and $m \in DR$ in Equation 11 turn out to be equal in this instance.

We have to note that, while this metric favours coverage-based resistant mutants (see Definition 7) and difficult to reach mutants (see Definition 9), coverage-based resistant hard to kill mutants receive the highest value (see Definition 8).

At this point, we can define the new metrics for a mutation operator derived from QC_m, analogously to the metrics defined from Q_m (see Section 2.5):

Definition 11 (Coverage-based quality of a mutation operator)

$$QC_o(M_o, T_o) = \frac{1}{|M_o|} \sum_{m \in M_o} QC_m(M_o, T_o) \tag{12}$$

Definition 12 (Coverage-based quality of the dead mutants) generated by a mutation operator

$$QC_{D_o}(D_o, T_o) = \frac{1}{|D_o|} \sum_{m \in D_o} QC_m(D_o, T_o) \tag{13}$$

QC_O and QC_{D_O} are similar to Q_O and Q_{D_O} respectively, but replacing Q_m with QC_m in their formal definition. Please notice that the minimisation of the test suite is a desirable property for the reliability of the results when computing these metrics, as we will discuss later in the paper.

As an example of the coverage-based quality metric, consider the mutant m_4 from the matrices EM and CM (see Section 4.2), where $|M| = 8$, $|E| = 1$, $|T| = 5$. We need to obtain the following values to calculate QC_{m_4}:

- $T_{m_4} = \{test_1, test_5\}$ (test cases reaching m_4).
- $|M_{test_1}| = 4$ (mutants reached by $test_1$).
- $|M_{test_5}| = 5$ (mutants reached by $test_5$).
- $|E_{test_1}| = 0$ (equivalent mutants reached by $test_1$).
- $|E_{test_5}| = 1$ (m_8 is an equivalent mutant).
- $|N_{test_1}| = |M_{test_1}| - |E_{test_1}| = 4 - 0 = 4$.
- $|N_{test_5}| = |M_{test_5}| - |E_{test_5}| = 5 - 1 = 4$.
- $K_{m_4} = \{test_5\}$ (test cases killing m_4).
- $|C_{test_5}| = 2$ (mutants killed by $test_5$).

Supposing that $MCOV = 4$ (see Definition 9), then $m_4 \in D \setminus DR$, and Q_{m_4} is calculated as follows:

$$QC_{m_4} = 1 - 2/(4 + 4) = 0.75$$

Table 1 Q_m and QC_m calculations for matrix EM (Figure 3) and matrix CM (Figure 4)

	m_1	m_2	m_3	m_4	m_5	m_6	m_7	m_8
Q	0.86	0.86	0.91	0.94	0.97	0.97	0.94	0.00
QC	0.58	0.44	0.62	0.75	0.95	0.95	0.88	0.00

As an illustration of the difference between Q_m and QC_m, these metrics have been calculated with respect to the matrices in Figure 3 and 4. The values of both metrics are presented in Table 1. This table shows that m_4 and m_7 are equally valued by the original metric ($Q_{m_4} = Q_{m_7} = 0.94$), as commented in the previous section. On the contrary, as expected, QC_m indicates that m_7 is of a higher quality than m_4 ($QC_{m_7} > QC_{m_4}$).

We can also observe that:

- m_5 is a difficult to reach mutant, as it is only reached by $test_4$, which in turn only reaches three other non-equivalent mutants ($N_{test_4} = \{m_1, m_5, m_6\}$).
- m_6 is a coverage-based resistant mutant, as it is reached by all the test cases, but it is only killed by $test_2$.

Recall that m_5 and m_6 are equally valuable for the refinement of the test suite, so they receive the same mark by QC_m ($QC_{m_5} = QC_{m_6} = 0.95$).

5 Empirical Evaluation

5.1 Research Questions

As mentioned in the introduction, the aim of this evaluation is to assess the performance of the coverage-based quality metric in comparison with the original metric. The application of the coverage-aware metric to C++ class-based operators serves this particular purpose. Therefore, we aim to answer the following research questions:

- **RQ1: How does the coverage information alter the quality assigned to each mutation operator?** We intend to know the value that the metric assigns to each operator taking into account the coverage of the test suite, and observe the differences with the original metric. Then, it will be interesting to study the distribution of mutants in relation to the value attached to each operator, including difficult to reach mutants as it is currently unknown the impact of these mutants on the quality assigned to mutation operators.
- **RQ2: Does the coverage-based quality metric outperform the original quality metric in terms of test suite size?** We aim to apply test-quality selective mutation through the operator classification obtained with both the original and the coverage-based metric, and then to study if the new metric allows us to reduce the test cases that would be lost when some of the operators are excluded in comparison with its original counterpart.
- **RQ3: Does the coverage-based metric allow operators to be discarded without losing high-quality test cases when compared to the original metric?** Applying the same selective strategy as in the previous question, it will be interesting to analyse the specificity of the test cases contained in the test suite. When some of the operators are removed,

this evaluation will complement the study by reporting whether we are losing test cases killing few or many mutants. A suitable metric should favour those mutants leading to the design of specific test cases which might only be created through the examination of few mutants.

Table 2 List of Class Mutation Operators Used in These Experiments

Group	Operator	Description
Inheritance	**IHD**	Hiding variable deletion
	IHI	Hiding variable insertion
	ISD	Base keyword deletion
	ISI	Base keyword insertion
	IOD	Overriding method deletion
	IOP	Overriding method calling position change
	IOR	Overriding method rename
	IPC	Explicit call of a parent's constructor deletion
Polymorphism and dynamic binding	**PCI**	Type cast operator insertion
	PMD	Member variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PNC	New method call with child class type
Method overloading	**OMD**	Overloading method deletion
	OMR	Overloading method contents replace
	OAN	Argument number change
Exception handling	**EHC**	Exception handling change
Object and member replacement	**MCO**	Member call from another object
	MCI	Member call from another inherited class
Miscellany	**CTD**	*this* keyword deletion
	CTI	*this* keyword insertion
	CID	Member variable initialization deletion
	CDC	Default constructor creation
	CDD	Destructor method deletion
	CCA	Copy constructor and assignment operator overloading deletion

5.2 Class Mutation Operators for C++

Class-based operators for this important industrial-strength programming language were recently defined by Delgado-Pérez et al. (2015). The set of class operators for C++ encompassed both adapted operators for Java (Ma et al., 2002)/C# (Derezińska, 2006) and tailored operators to C++ features. A revised list of these operators has been implemented in the mutation tool called *MuCPP*[2], where they underwent a thorough study to increase the ratio of useful mutants generated (Delgado-Pérez et al., 2017a). This set of C++ class operators, which was previously assessed following a selective mutation approach (Delgado-Pérez et al., 2017b), will be analysed in the conducted experiments in the next section.

[2] https://ucase.uca.es/mucpp

Table 2 shows a description of the class operators that generate mutants in our experiments. Further details and examples about them can be found in previous papers (Delgado-Pérez et al. (2015) and Delgado-Pérez et al. (2017a)).

5.3 Case Studies and Test Suites

We carried out the experimental procedures explained in the next sections on a set of C++ open-source programs, namely[3]:

- *Matrix TCL Pro* (*Tcl*) (Matrix TCL Pro, 2017): a library to perform matrix algebra calculations (LOC: 3,228; $|T| = 24$).
- *XmlRpc++* (*Rpc*) (XmlRPC, 2017): a library that implements the XML-RPC protocol to incorporate client-server communication through HTTP support into other C++ programs (LOC: 2,194; $|T| = 34$).
- *Dolphin* (*Dph*) (Dolphin, 2017): the default navigational file manager in KDE desktop applications (LOC: 3,667; $|T| = 70$).
- *Tinyxml2* (*Txm*) (Tinyxml2, 2017): a lightweight and efficient XML parser that can be integrated into C++ applications (LOC: 2,620; $|T| = 62$).
- *KMyMoney* (*Kmy*) (KMyMoney, 2017): a KDE desktop application for personal finance management (LOC: 13,709; $|T| = 247$).
- *QtDom* (*Dom*) (QtDOM, 2017): a module of the known Qt framework that provides a C++ implementation of the DOM standard (LOC: 2,117; $|T| = 56$).

Taking into account that adequate test suites are required to calculate the quality metric, these programs were selected for being accompanied by non-trivial test suites. Thus, starting from the test suite distributed with the aforementioned programs, we manually extended the test suite with new test cases to kill surviving mutants until reaching an adequate test suite for each of these SUTs (the final size of the test suite is shown above along with each case study). We could not however classify some of the mutants as equivalent because we were unable to ascertain this condition with high confidence. For those mutants, we used the term *undecided* (Segura et al., 2011) to avoid skewing of results when computing the metrics.

Tables 3-8 show, per each SUT and mutation operator, the total number of mutants generated and their classification into dead, equivalent and undecided.

To achieve the minimality of the test suite in our experiments, we used the same exact algorithm applied by Estero-Botaro et al. (2015) in their study. In this regard, we have to note that the minimisation of the test suite is performed using the execution matrix. Still, the same columns that disappear from the execution matrix when minimising the test suite have to be removed from the coverage matrix as well.

[3] Further details about the programs under study can be found in the paper by Delgado-Pérez et al. (2017b).

Table 3 Classification of Mutants in TCLPro

Operator	**Total**	**Dead**	**Equivalent**	**Undecided**
OMD	46	38	8	0
OMR	34	33	1	0
MCO	3	3	0	0
CID	40	38	2	0
CDD	2	0	2	0
CCA	10	3	7	0
Total	135	115	20	0

Table 4 Classification of Mutants in XmlRpc++

Operator	**Total**	**Dead**	**Equivalent**	**Undecided**
IHI	4	2	2	0
ISD	1	1	0	0
ISI	3	2	1	0
IOD	3	1	2	0
IOR	15	0	15	0
IPC	1	1	0	0
PCI	3	2	1	0
PPD	1	0	1	0
OMD	10	9	1	0
OMR	10	10	0	0
MCO	48	38	10	0
EHC	2	1	1	0
CID	17	14	3	0
CDC	2	2	0	0
CDD	5	2	3	0
CCA	2	2	0	0
Total	127	87	40	0

5.4 Coverage Information

The data required for the calculation of the coverage-based metric was collected as follows:

1. Each test case was independently run against the original program. In these executions, we used the application *gcov*[4] to get a report of the lines reached by each test case.
2. We instrumented the mutation tool *MuCPP* to retrieve the lines where the mutations were injected, reusing the *Clang* libraries to this end.
3. Finally, we obtained the coverage matrix by automatically matching the lines of the code mutated with the lines reached by each test case.

We can broadly classify the mutation operators with regard to the coverage depending on the mutated element:

– **Statement**: When the mutation only affects a concrete statement, we simply retrieve the line containing the statement.

[4] https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

Table 5 Classification of Mutants in Dolphin

Operator	**Total**	**Dead**	**Equivalent**	**Undecided**
ISI	11	9	2	0
IOD	18	15	3	0
IOR	30	3	27	0
IPC	5	2	3	0
OMD	3	2	1	0
OMR	6	5	1	0
MCO	85	68	7	10
CTI	2	2	0	0
CID	41	23	18	0
CDD	2	0	2	0
CCA	5	0	5	0
Total	208	129	69	10

Table 6 Classification of Mutants in Tinyxml2

Operator	**Total**	**Dead**	**Equivalent**	**Undecided**
IHI	47	41	6	0
IOD	25	24	1	0
IOP	8	8	0	0
IOR	11	10	1	0
PCI	190	138	20	32
PMD	3	0	3	0
PPD	7	5	2	0
OMD	37	23	14	0
MCO	19	18	1	0
MCI	39	13	26	0
CID	34	24	10	0
CDC	3	3	0	0
CDD	6	3	3	0
CCA	4	0	4	0
Total	433	310	91	32

- **Block**: Several operators mutate a block of code that comprises various lines of code, such as the deletion of a method or an exception handler. In these cases, we use the first line of the block as a reference to know whether a test case reaches the mutant.
- **Structure of a declaration:** When the mutated element changes the structure of a declaration, the coverage information has to be obtained as a combination of the different lines where the declared element is referenced. Therefore, a test case executing at least one of the collected lines reaches the mutant.

We also developed a checker that verifies the concordance of the data included in the execution and the coverage matrix. As mentioned in Section 4.4, we need to check that, whenever a mutant is killed by a test case, that mutant has been reached.

Table 7 Classification of Mutants in KMyMoney

Operator	**Total**	**Dead**	**Equivalent**	**Undecided**
IHD	1	1	0	0
IHI	23	8	15	0
ISI	3	0	3	0
IOD	1	1	0	0
IPC	18	12	6	0
PCI	15	14	1	0
PMD	1	0	1	0
PPD	18	4	14	0
OMD	13	9	4	0
OMR	32	32	0	0
OAN	7	3	4	0
MCO	87	76	7	4
EHC	6	1	5	0
CID	48	26	22	0
CDC	5	4	1	0
CDD	4	2	2	0
CCA	2	0	2	0
Total	284	193	87	4

Table 8 Classification of Mutants in QtDom

Operator	**Total**	**Dead**	**Equivalent**	**Undecided**
IHI	46	21	25	0
ISI	2	1	1	0
IOD	32	28	2	2
IOP	2	0	2	0
IOR	1	0	1	0
IPC	8	8	0	0
PCI	451	293	155	3
PMD	4	0	4	0
PPD	16	2	12	2
PNC	2	2	0	0
OMD	22	16	6	0
OMR	16	16	0	0
MCO	43	36	7	0
CTD	1	1	0	0
CTI	1	1	0	0
CID	16	6	9	1
CDD	4	0	4	0
CCA	14	4	8	2
Total	681	435	236	10

5.5 Experimental Setup

We first calculated both the original and the coverage-based quality metric in the case studies listed in Section 5.3. The authors of the original quality metric established a threshold of four mutants as the minimum number of mutants that a mutation operator should generate so that the value of the metric was

significant. We complied with this restriction in both metrics for consistency with the experiments performed in that paper.

As for difficult to reach mutants, we carried out a sensitivity analysis with our SUTs to set the value of $MCOV$ (see Definition 9). We calculated the coverage-based quality metric of each mutation operator applicable to each program using different thresholds for MCOV (from 2 to 6)[5]. After studying the results, we concluded that the value $MCOV = 4$ is an appropriate threshold to avoid overfitting the results to these programs.

Once we had the values for each case study and operator, we compared the results of the original and the coverage-based metric by applying a selective process based on the values of each metric. Recall that we calculate the *test suite specificity* in addition to its *size* so that, in case of losing some test cases, we can determine whether those test cases are or not high-quality test cases (see Section 3.3 for further details). To that end, we followed the below steps for each metric and case study:

1. Calculate an adequate and minimal test suite for the whole set of operators.
2. Sort the mutation operators that could be graded with the metric. The operator with the highest mark is placed at the top of the classification.
3. Apply test-quality selective mutation:
   (a) Remove the operator at the bottom of the classification.
   (b) Compute an adequate and minimal test suite for the remaining operators.
   (c) Calculate (see details of these measurements in Section 3.3):
      – **Size:** Percentage of test cases loss when compared to the original adequate and minimal test suite obtained in step 1. Recall, as we focus on the improvement of the test suite and not in its mutant adequacy, we compute the percentage of test cases loss instead of the mutation score.
      – **Specificity:** Average percentage of mutants killed by the test cases.
4. Repeat step 3 as many times as operators there are in the classification (except for the operator in the top).

Finally, we compared the results between using Q_o and QC_o with the same number of operators.

As a final point, it is worth mentioning the strategy to handle with draws among operators in step 2. In order of priority:

1. Using the sort of operators that provides the best result in terms of lost test cases.
2. Sorting operators by the number of mutants generated, where the most prolific is placed at the bottom.
3. Otherwise, operators are sorted in the same order as they are shown in Tables 3-8.

[5] The results of this analysis can be found in:
https://ucase.uca.es/coverage-based-metric

6 Assessing the Performance of the Coverage-Based Quality Metric

6.1 Analysis of the Quality Metric Values

Table 9 presents the value of the original quality metric of each operator (Q_o) in the analysed SUTs[6]. As mentioned in Section 5.5, the metric is not measured for those operators generating fewer than four mutants in a case study. We have represented these cases with the symbol '-' in this table. Analogously, Table 10 collects the values of the coverage-based quality metric (QC_o) in each operator and case study. As we calculate this metric under the same conditions as the original metric, we obtain a value for the same operators in each program.

When comparing the results of the coverage-based metric (Table 10) with the ones obtained by the original metric (Table 9), we can observe a general reduction in the values. This reduction is however the expected outcome since the denominator in the formula of QC_m (Equation 11) is greater or equal than the one to compute Q_m (Equation 1). That is, the coverage-based metric does not use the size of the execution matrix for the calculations, but only the number of 'x' in the coverage matrix, which diminishes the values of the metric.

We can observe that some operators like *IOD*, *OMD* and *OMR* usually obtain good marks in both metrics. On the contrary, some operators like *PPD*, *CDD* or *CCA* are not well valued by these metrics. However, there are also notable differences between the value assigned by both metrics. For instance, the operator *IPC*, which is the third best-valued operator following the original metric in *Dom*, falls to the sixth position according to the coverage-based metric. As such, while the operator ranking based on these metrics is similar in some SUTs like *Dph*, this classification turns out to be quite different in other programs, such as *Tcl* or *Txm*.

Unlike traditional operators, most class mutation operators are not applied in every SUT and they depend on the object-oriented features used by the programmer. As a result, only five operators create mutants in all the SUTs (*OMD*, *MCO*, *CID*, *CDD* and *CCA*). Furthermore, this type of operator usually engenders a low number of mutants, so finally the quality metric can only be measured for every case study in one (*CID*) out of those five operators. The existence of several operators with the value 0 is due to the number of equivalent mutants or the few mutants that they generate. When the operator produces few mutants, it is more likely that only one test case suffices to kill all the mutants from that operator; this always results in the lowest value for the quality metric. This fact can be seen in Table 11, which shows the number of dead mutants generated by the mutation operators when they are sorted through the metrics in each case study. As it can be seen, both metrics tend to gather the operators generating more mutants in the top positions of the

[6] Some of the values presented in this table slightly differ from the results in the previous work by Delgado-Pérez et al. (2017b) because of some changes in the test suites used.

Table 9 Value of the Quality Metric of the Mutation Operators (Q_o) in each Case Study

Oper.	**Tcl**	**Rpc**	**Dph**	**Txm**	**Kmy**	**Dom**
IHI		0		0.73	0.29	0.39
ISI		-	0.57		-	-
IOD		-	0.80	0.78	-	0.89
IOP				0		-
IOR		0	0.07	0.65		-
IPC		-	0.30		0.65	0.79
PCI		-		0.71	0	0.60
PMD				-	-	0
PPD		-		0	0.17	0.11
OMD	0.80	0.82	-	0.57	0.68	0.71
OMR	0.88	0.92	0		0.98	0.89
OAN					0.31	
MCO	-	0.74	0.79	0.72	0.89	0.73
MCI				0.30		
EHC		-			0	
CID	0.83	0.74	0.52	0.55	0.52	0.29
CDC		-		-	0.47	
CDD	-	0.30	-	0	0	0
CCA	0.17	-	0	0	-	0.25

Table 10 Value of the Coverage-Based Quality Metric of the Mutation Operators (QC_o) in each Case Study

Oper.	**Tcl**	**Rpc**	**Dph**	**Txm**	**Kmy**	**Dom**
IHI		0		0.58	0.28	0.34
ISI		-	0.51		-	-
IOD		-	0.68	0.31	-	0.49
IOP				0		-
IOR		0	0.07	0.21		-
IPC		-	0.20		0.58	0.29
PCI		-		0.25	0	0.26
PMD				-	-	0
PPD		-		0	0.11	0.07
OMD	0.17	0.35	-	0.05	0.58	0.35
OMR	0.08	0.43	0		0.61	0.38
OAN					0.19	
MCO	-	0.46	0.78	0.09	0.05	0.54
MCI				0.20		
EHC		-			0	
CID	0.11	0.38	0.26	0.53	0.46	0.18
CDC		-		-	0.28	
CDD	-	0.30	-	0	0	0
CCA	0.14	-	0	0	-	0.17

classification. *Txm* is the best example of this fact, where all the operators creating a number of mutants greater or equal than 10 are above the rest of operators producing fewer mutants. These results suggest that there is a link between the quality metrics under study and the number of dead mutants in each operator. Nevertheless, whether the number of mutants is more or less

Table 11 Number of Dead Mutants Generated by each Mutation Operator, Sorted by the Original (Q) and the Coverage-Based (QC) Quality Metric in each Case Study (see Tables 9 and 10 respectively). For instance, the First Operator in *TCL* according to Q is *OMR*, which Generates 33 Mutants; according to QC, the First Operator is *OMD* with 38 Mutants.

Position	***Tcl***		***Rpc***		***Dph***		***Txm***		***Kmy***		***Dom***	
	Q	QC	Q	QC	Q	QC	Q	QC	Q	QC	Q	QC
1	33	38	10	38	15	68	24	41	32	32	28	36
2	38	**3**	9	10	68	15	41	24	76	9	16	28
3	38	38	38	14	9	9	18	24	9	12	8	16
4	3	33	14	9	23	23	138	138	12	26	36	16
5			2	2	2	2	10	10	26	8	16	21
6			2	2	3	3	23	13	4	4	293	8
7			0	0	0	0	24	18	3	3	21	293
8					5	5	13	23	8	4	5	5
9							0	0	4	**76**	4	4
10							3	3	14	14	2	2
11							5	5	1	1	0	0
12							8	8	2	2	0	0

high does not seem a decisive factor in the operator classification. While the aforementioned fact applies to both metrics, there are two cases that suggest that the new metric is not so linked to the number of mutants as the original metric (marked in bold in Table 11). In *Tcl*, the operator *CCA*, generating only 3 dead mutants, is able to surpass the operators *CID* and *OMR*, each of them generating more than 30 mutants. Also the operator *MCO* in *Kmy*, which is the most prolific in that SUT with 76 dead mutants, is quite lower in the coverage-based operator ranking (9th position) than in the ranking based on the original metric (2nd position). That is indeed the biggest difference between metrics in these case studies.

Finally, this study requires an evaluation of the impact of difficult to reach mutants on the quality assigned to each operator, as they are treated as a separate case by the coverage-based metric. The appearance of difficult to reach mutants mostly depends on where in the code the operator injects the mutation. By their nature, some operators may be more prone to produce this kind of mutants than others. For instance, it is likely that only a few test cases invoke a concrete overloaded method or a particular constructor, and this fact could affect operators which target such methods. In other cases, as the operator *CDD* in these experiments, the dead mutants are considered difficult to reach because of a lack of coverage data (i.e., $\sum_{t \in T} |N_t| \leq MCOV$, as explained in Definition 9). Given this context, it is important to clarify whether producing a high percentage of difficult to reach mutant is a decisive factor in the quality attached to each operator. With this aim, Figure 5 shows the percentage of difficult to reach mutants generated by each operator in each case study when the mutation operators are sorted according to their quality. As it can be seen, any pattern can be established between the order of the operators and the percentage of difficult to reach mutants that they produce, given that the highest percentages are spread across the figure. For instance, while the operator *MCO* does not generate any difficult to reach mutant in

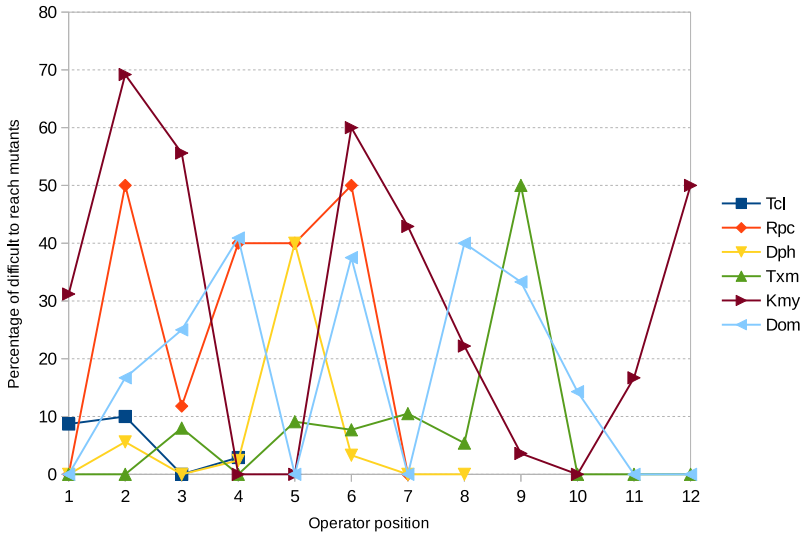

Fig. 5 Percentage of Difficult to Reach Mutants Generated by each Mutation Operator, Sorted by the Coverage-Based Quality Metric in each Case Study. The Leftmost and the Rightmost Point Represent respectively the Best-Valued and Worst-Valued Operator in each Case Study (see Table 10). For instance, 50% of the Mutants Generated by the Operator *OMR*, the Second Most-Valued Operator in *RPC* ($QC_o = 0.43$), are Difficult to Reach.

Dom and is in the first position of that operator classification, this operator is responsible for the second highest percentage of this kind of mutants in *Txm* and it is however in the middle of that ranking (7th position). Hence, we conclude that the operator classification based on this metric does not directly result from the percentage of difficult to reach mutants in this study.

6.2 Test Suite Size Analysis

Table 12 shows the results of applying test-quality selective mutation based on the original (Q) and the coverage-based metric (QC) in terms of test suite size. These values represent the percentage of test cases that are missing from the original adequate and minimal test suite when the operators under that row are removed (the mutants from those operators are discarded). As an example, in the case of *Rpc*, only using the first four operators following the original metric (*OMR*, *OMD*, *CID* and *MCO*), we will be assuming a loss of 26.7% of the test cases. We have highlighted in bold the cases when the coverage-based metric leads to a better result than the original metric, and we have used the symbol '=' when the outcome is the same.

To begin with, the selective strategy following the metrics yields good results, validating the main approach of these metrics regarding the quality. The results in *Dph*, *Txm* and *Dom* are the most notable; in these SUTs, none of the test cases are lost after discarding several operators. It is also remarkable

that only applying the top three operators, we are able to maintain on average 69.0% and 70.8% of the test cases in the adequate and minimal test suite using the original and the coverage-based metric respectively. We have to note that, in the case of *Rpc* and *Kmy*, the percentage of test cases loss before starting the selective process is respectively 20% and 2.9% (instead of 0%) because of the operators that generate fewer than four mutants; these operators cannot be measured by the metric and are previously removed, but they produce some useful mutants required to maintain the same adequate and minimal test suite in these cases.

When comparing both metrics, we can observe that the results are positive for the coverage-based metric in *Tcl*, *Rpc*, *Dph* and *Txm*, and negative only in *Kmy*. In the case of *Dom*, the result varies depending on the number of operators excluded. The behaviour of QC_o is especially satisfactory in *Txm*, where this metric outperforms the original one in 5 out of 11 subsets of operators and there is a tie in other 4 cases. In the previous section, we could see that the operator *CCA* climbed to the second position in the classification by the coverage-based metric in *Tcl*; even when this operator only generates 3 mutants, that ranking turns out to be better in comparison with the original metric. Indeed, just selecting the two first operators based on the coverage-based metric, we would generate 41 mutants and the loss would be 26.7%, while we would generate 71 mutants and the loss would be 40% taking into account the original metric.

In the light of the results from Table 12, taking into consideration the coverage information of the test suite is a fruitful approach for measuring the quality of mutation operators. The metric, however, assigns a low value to the operator *MCO* in *Kmy*, which results in a significant loss of test cases when that operator is excluded in the selective process (row 8 in Table 12). We have no evidence so far about the nature of the test cases that *MCO* could help design to extend a test suite for this program. To that end, in the next section we carry out the analysis of the test suite specificity to complement this study.

6.3 Test Suite Specificity Analysis

Table 13 depicts a comparison in terms of specificity between the test suite formed with Q and QC when performing the selective process explained in Section 5.5. Each value of this table represents the average percentage of mutants killed by the test cases included in the adequate and minimal test suite for the operators that still remain in the subset. The format of this table is similar to Table 12. As an example, only with four operators in *Kmy*, the percentage of mutants killed by the test cases obtained through the coverage-based metric is 7.47% on average, and 7.70% in the case of the original metric; thus, in this example we say that the coverage-based metric outperforms its original counterpart because the former leads to produce mutants killed by few test cases (and, therefore, mutants with potential to guide the tester on the refinement of that test suite with specific test cases). We should note that

Table 12 Comparison of the Percentage of Test Cases Loss in each SUT when Applying Test-Quality Selective Mutation Based on the Original (Q) and the Coverage-Based (QC) Quality Metric (# Op. = number of operators selected)

# Op.	*Tcl*		*Rpc*		*Dph*		*Txm*		*Kmy*		*Dom*	
	Q	QC	Q	QC	Q	QC	Q	QC	Q	QC	Q	QC
1	46.7	**26.7**	60.0	=	59.1	**45.4**	76.5	82.3	60.0	=	64.0	80.0
2	40.0	**26.7**	60.0	**33.3**	22.7	=	64.7	**58.8**	45.7	=	40.0	56.0
3	0	6.7	33.3	**26.7**	22.7	=	58.8	**47.1**	31.4	40.0	40.0	**32.0**
4	0	=	26.7	=	4.6	=	41.2	**29.4**	22.9	31.4	32.0	**20.0**
5			20.0	=	0	=	35.3	**23.5**	14.3	28.6	20.0	=
6			20.0	=	0	=	23.5	**17.6**	14.3	28.6	0	20.0
7			20.0	=	0	=	5.9	11.8	11.4	25.7	0	=
8					0	=	0	=	8.6	25.7	0	=
9							0	=	8.6	=	0	=
10							0	=	5.7	=	0	=
11							0	=	2.9	=	0	=
12							0	=	2.9	=	0	=

Table 13 Comparison of the Average Percentage of Killed Mutants by the Test Cases in each SUT when Applying Test-Quality Selective Mutation Based on the Original (Q) and the Coverage-Based (QC) Quality Metric (# Op. = number of operators selected)

# Op.	*Tcl*		*Rpc*		*Dph*		*Txm*		*Kmy*		*Dom*	
	Q	QC	Q	QC	Q	QC	Q	QC	Q	QC	Q	QC
1	21.52	**20.71**	22.80	**21.65**	16.54	22.35	40.4	**26.45**	5.81	=	17.16	19.31
2	20.19	20.71	21.07	22.53	18.29	=	33.44	**29.91**	7.17	7.42	15.13	17.18
3	18.26	18.32	19.31	20.69	18.88	=	33.87	**32.11**	8.18	**7.70**	15.13	15.38
4	18.26	=	18.29	=	16.46	=	33.23	**32.02**	7.70	**7.47**	15.38	**15.24**
5			18.29	=	16.46	=	32.20	**31.24**	7.98	**7.83**	15.24	=
6			19.73	=	15.82	=	33.05	**31.36**	7.96	**7.83**	16.95	**15.24**
7			19.73	=	15.82	=	32.38	**31.70**	7.87	**7.67**	16.95	=
8					15.82	=	32.41	=	7.66	7.67	16.95	=
9							32.41	=	7.66	=	16.95	=
10							32.41	=	7.91	=	16.95	=
11							32.41	=	7.99	=	16.95	=
12							32.41	=	7.99	=	16.95	=

the difference in percentage among the SUTs is due to the test suite used for each program.

By matching the results in this table with the loss of test cases in Table 12, we can gain a deeper understanding of the performance of the metrics:

- **Xml and Dph**: In the cases when QC_o outperforms Q_o in terms of size, Q_o outperforms QC_o in terms of specificity. In such cases, the test cases acquired by means of QC_o when compared to Q_o are not so specific. However, in the first row in *Xml* where the metrics are tied in the size aspect, now QC_o outperforms Q_o.
- **Kmy**: QC_o is better in the specificity aspect, but worse in the size aspect. That means that the test cases lost by using QC_o when compared to Q_o are not actually specific (in other words, those missing test cases seem mostly straightforward as they kill many mutants). Therefore, our metric is prone to maintain interesting test cases.
- **Txm**: QC_o is better than Q_o in both the size and specificity aspect. That means that Q_o not only is losing more test cases, but also that those test cases are interesting to retain.

In general, these results coupled with the results of the test suite size analysis suggests that the coverage-based metric provides better performance than the original metric.

6.4 Answers to Research Questions

Answer to RQ1: How does the coverage information alter the quality assigned to each mutation operator?

The use of the coverage matrix implies having more precise information available for the calculations and, therefore, the values that the coverage-based quality metric gives to mutation operators are in a different scale when compared to the original quality metric. However, while it is easy to observe that some operators are often well valued by both metrics, it is also clear that the new coverage-based metric alters the quality estimation of some mutation operators in several cases. Both quality metrics are likely to assign low values to operators producing few mutants, but this relation seems to be closer in the case of the original metric. Finally, the results show that the fact of generating difficult to reach mutants does not have significant impact on the mutation operator quality.

Answer to RQ2: Does the coverage-based quality metric outperform the original quality metric in terms of test suite size? Yes, the coverage-based metric does outperform the original metric in the conducted experiments, yielding better results in most of the cases (see Table 12). Additionally, it is remarkable that the removal of several operators barely reduced the effectiveness in most programs, which validates the use of a quality metric. In any event, this metric requires to be complemented with information

about test suite specificity in order to know whether operators with low quality mainly lead to the generation of specific or straightforward test cases.

Answer to RQ3: Does the coverage-based metric allow operators to be discarded without losing high-quality test cases when compared to the original metric? Yes. Although the original metric exhibits a good performance in this aspect, when we analyse the test suite size (Table 12) and the specificity (Table 13) together, the coverage-based metric has the potential to retain more specific test cases in the test suite when the worst rated operators are excluded. As a conclusion, the test cases lost when applying the selective strategy based on this metric usually coincide with those that are not so hard to design.

7 Threats to Validity

The results presented are subject to several limitations: some of them are inherent to mutation testing in general, while others are due to restrictions in the conducted studies.

Construct validity: Regarding the construct validity, we have not only considered the test suite size to measure the performance of the quality metrics, but also its specificity. In the former case, we have seen that the size can be affected by the fact that some trivial test cases might be missing, so computing the specificity helps complement this measurement. In the case of the specificity, we have measured the average of mutants killed by the test cases, but the quality of a test suite might be measured through other variables.

Internal validity: The process to obtain execution and coverage matrices and the experimental results reported in this paper has been automated wherever possible, but there may exist errors in the scripts implemented and tools used. The results might also be influenced by the manual determination of equivalent mutants; to counter the threat that the quality metric punishes mutants incorrectly classified as equivalent, we have categorised as undecided instead of as equivalent those mutants for which we were unsure. As we cannot isolate the measure of quality from the test suite, the mutant-driven design of new test cases to achieve the necessary adequacy of the test suite and the use of a single test suite can also impact the results. In this case, using the test suite distributed with the programs as the model to generate new test cases and minimising the test suite with an exact algorithm to exclude unproductive test cases, alleviate respectively the effect of these potential threats to the validity of the results.

External validity: Most class mutation operators often generate no or few mutants in each class because they can only be applied under specific circumstances. This fact hindered us from measuring the quality of several operators. As a result, only a subset of operators could be studied in each SUT, but we have used six programs with different features and sizes to minimise this threat. The quality metrics were applied to class mutation operators in C++, so we cannot know if the results hold in other contexts.

8 Additional Considerations

Cost implications: As mentioned before, the coverage-based metric is to be used in advance and not directly in practice. That is, we do not implement the metric in the mutation tool but we make use of the results of previous research to know the most valuable operators instead. As such, applying the quality metric does not involve any additional cost in the testing activity (on the contrary, it is only useful to reduce its cost). For the same reason, we obtain an adequate and minimal test suite to calculate the quality metric but not in practice.

Test suite size: In testing, it is usually desirable to reduce the number of test cases that must be run as much as possible. Reducing the test suite size is a good idea when the test quality is not penalised. In other words, we should not reduce the cost at the expense of the quality. However, in test-quality selective mutation we favour a subset of operators that is likely to guide us to a larger test suite than others. Given that the goal of the quality metric is to highlight operators with great potential in help us add high-quality test cases, it is worthwhile to form a test suite with as many test cases as possible to increase the confidence in its fault detection capability.

Operator classification: The final goal of using the coverage-based quality metric is to sort mutation operators by their associated quality metric in a general ranking. However, we have seen that there might be variance in the position of the operators for different programs. More studies are required to analyse this further, especially applying this quality metric to more applications in order to achieve an operator classification as generalisable as possible (also addressing the whole set of mutation operators defined for the language).

Cost reduction techniques: Test-quality selective mutation has been applied to analyse the performance of the proposed quality metric in this work. However, it was out of the scope of this paper to evaluate the effectiveness of a selective strategy in comparison with other techniques for the reduction of mutants, or to assess if a selective strategy is appropriate for this particular case study (C++ class-level operators). This aspect would merit additional studies in the future.

Language-independent metric: Although class mutation operators for C++ have been analysed in this paper as a case study to show the behaviour of the new metric, we should remark that the metric can be applied to any programming languages and domains.

9 Related Work

Coverage

Coverage information is usually collected in different mutation tools to reduce the computational cost of applying mutation testing (Vincenzi et al., 2006; Schuler & Zeller, 2009; Derezińska & Szustek, 2012; Mateo & Usaola, 2012;

Derezinska & Halas, 2014), but we used this information to devise a new metric to measure operator effectiveness. Bacterio (Mateo & Usaola, 2012) retrieves coverage data to prevent unnecessary executions of mutants generated with both traditional and class operators for Java. However, they report that, because of the use of mutant schemata, they could not leverage this information for some of their class operators (Mateo & Usaola, 2015). When a mutation operator changes in some way the structure of a declaration, the mutated elements are not directly executed and the coverage cannot be captured. This is the case of the operator *IHI*, which inserts in a subclass a variable member that hides another variable member with the same name in a base class. Since *MuCPP* creates a new version of the program for each mutation, we were able to locate the lines where the affected element was being referenced in the code. We used the tool *gcov* to know which test cases reached the statements where the mutations were injected. In the work by Papadakis & Malevris (2011), they analysed whether a test case is able to infect the state of the system or not. Thus, following their approach would theoretically result in a more accurate measurement of the coverage-based quality metric.

In their study about the correlation between coverage and test suite effectiveness, Inozemtseva & Holmes (2014) defined the normalized effectiveness measurement as the number of mutants detected by a test suite, divided by the number of non-equivalent mutants that the test suite actually covers. The denominator in that metric is more precise than the one used by the mutation score (the total number of non-equivalent mutants). The coverage-based metric is even more accurate regarding the coverage, as the coverage of the test suite is calculated independently for each mutant based on the number of test cases reaching it and the mutants reached by those test cases.

There are several other works related to mutation testing analysing code coverage. Zhang et al. (2012) developed the tool *ReMT* with a focus on regression mutation testing. By studying the coverage in the old version of the program, this tool can safely reuse previous results and reduce the number of mutants. Coverage also helps *ReMT* prioritise test cases for each mutant. Papadakis et al. (2014) recently analysed the impact on the code coverage of mutations in order to classify mutants and mitigate the effects of equivalence. Thanks to the coverage facilities added to *MutPy* to reduce the mutants generated and test cases executed, Derezinska & Halas (2014) were able to discuss the impact of code coverage in mutation testing. They concluded that code coverage can significantly decrease the overall mutation time both in first and higher order mutants.

Metrics

In the study by Estero-Botaro et al. (2015), they first calculated the number of equivalent, trivial and resistant mutants that each operator generated. Then, they computed the quality of each mutation operator trying to establish a threshold that allowed to discard some operators with a low quality. In our

paper, we tried to produce evidence that the quality metric is, in fact, a good indicator of the mutants that can really help improve the test suite with high-quality test cases. To that end, we put into practice a selective approach for the removal of operators depending on the quality that they exhibited. Estero-Botaro et al. (2015) analysed in depth the connection between their quality metric and the formulas of effectiveness by Derezińska (2006), utility of operators by Smith & Williams (2009) and mutation operator strength by Hu et al. (2011). In this paper, we also studied the conditions under which our metric behaves as the original metric. Stubborn mutants defined by Yao et al. (2014) (non-equivalent mutant not killed by a test suite complying with branch coverage criteria) are less resistant to be killed than coverage-based resistant mutants since stubborn mutants only require that a single test case reaching the mutation is not able to uncover it. The mutation tool MuRanker (Namin et al., 2015) outputs a particular ranking of mutants for each SUT based on a prediction about how difficult it is to kill each of its mutants. This approach is different from the use of a quality metric: the latter aims to reduce testing effort based on empirical evidence of the usefulness of each mutation operator.

While Derezińska (2006) and Smith & Williams (2009) did not impose any restriction to the test suite when computing their respective metrics, Mresa & Bottaci (1999) and Hu et al. (2011) assessed operators with adequate and non-redundant test suites. Estero-Botaro et al. (2015) went further by establishing the condition of minimality to the test suite. We also derived adequate and non-redundant test suites of the minimum size because they exclude test cases that may cause deviations in the values. Ammann et al. (2014) recently proposed using minimal sets of mutants in addition to minimal test suites, but they used approximation algorithms instead of an exact version. From their experimental results, the authors of the original quality metric suggested that the operators with an average metric below the medians of Q_o and Q_{D_o} may be candidates to be discarded in the future. This proposal would require further consideration with the new quality metrics. Similarly, Hu et al. (2011) recommended the omission of those expensive class operators in Java with a low mutation operator strength at the same time.

Selective mutation

Selective mutation has been broadly used in the past as a way to reduce the large computation expenses. In the first study about this cost reduction technique, Wong & Mathur (1993) found that just using two operators provided a significant reduction in the number of mutants examined and test cases required, but this only implied a small loss in the effectiveness. The experiments conducted by Offutt et al. (1993), excluding the six most prolific operators for Fortran, yielded a reduction over 60% of the mutants without a meaningful decrease of the mutation score. Removing operators of a similar syntactic category has been another traditional approach to selective mutation (Offutt et al., 1996; Mresa & Bottaci, 1999). Barbosa et al. (2001) and Namin et al.

(2008) tried to find sufficient sets of operators for C programs by defining a set of guidelines and a statistical analysis procedure respectively. Banzi et al. (2012) also applied a genetic algorithm for the selection of mutation operators. They used a multi-objective approach: maximise the mutation score and minimise the number of mutants generated. In this paper, we applied the traditional approach to selective mutation of removing all the mutants generated by the operators discarded. As mentioned before, other recent studies suggest that operator-based selection is not superior to mutant-based selection (Zhang et al., 2010; Gopinath et al., 2015). Delgado-Pérez et al. (Delgado-Pérez et al., 2017b), who proposed to assess the effectiveness of mutation operators for the evaluation and the refinement of the test suite separately, reported that rank-based selection is a preferable option to the removal of class-based operators in general.

Class mutation operators

Because of the increase in the presence of object orientation in the industry, many of the works related to mutation testing have focused on this paradigm (Ma et al., 2002; Derezińska, 2006). Most of the studies regarding class-based mutation operators have been carried out using mutation tools that implement Java (Ma et al., 2005) and C# operators (Derezińska & Szustek, 2007). Two of the most recent and relevant experiments about this kind of operators were conducted by Ma et al. (2009) and Derezińska & Rudnik (2012) for Java and C# respectively, and mutation operators at the class level were surveyed by Ahmed et al. (2010). Although there exists a previous work that proposes different class-based mutations for C++ (Derezińska, 2003), a set of class operators for this language was recently defined and analysed by Delgado-Pérez et al. (2015, 2017a).

10 Conclusion

This paper has presented a coverage-aware quality metric that allows for a more accurate measurement of operator effectiveness when refining test suites with high-quality test cases. Our metric led us to work with novel concepts, such as coverage matrix (representing the mutants executed by each test case) and difficult to reach mutants (those which are scarcely covered by the test suite). As a first important finding, our metric satisfied the expectations as a mechanism for the reduction of the number of mutants without a meaningful loss of effectiveness. The empirical results when using class mutation operators in C++ suggest that our quality metric outperforms the comparable metric by Estero-Botaro et al. (2015). By exploiting test-quality selective mutation (obtaining a reduced set of operators without significantly losing test improvement power), we could observe that our metric enables us to form not only a larger test suite, but also one that retains more specific test cases. The new

metric would also allow for a higher reduction of mutants since it is not so likely that the most prolific operators are among the most valued operators. Whether losing some effectiveness compensates the reduction in the cost or not, the decision should be taken by the tester depending on the thoroughness required with each SUT.

As for the future work, while this metric has shown a good performance when applied to C++ class-based operators, its evaluation with other mutation operators and programming languages could offer a deeper understanding as well as determine to which extent it is fruitful. Likewise, we aim at obtaining a ranking of operators based on this new metric as general as possible. This operator classification could serve as a practical reference to undertake test-quality selective mutation and balance the expense of the technique and the effectiveness when improving a test suite. Further refinements of this metric, or the definition of similar ones including new information not contemplated yet, could help us gain insight about operator effectiveness.

Acknowledgements This paper was funded by the Spanish Ministry of Economy and Competitiveness (National Program for Research, Development and Innovation), through the project DArDOS (TIN2015-65845-C3-3-R (MINECO/FEDER)) and the Excellence Network SEBASENet (TIN2015-71841-REDT), and by the research scholarship PU-EPIF-FPI-PPI-BC 2012-037 of the University of Cádiz. We also thank Francisco Palomo-Lozano for assisting us in using his algorithm to find minimal test suites.

References

Ahmed, Z., Zahoor, M., & Younas, I. (2010). Mutation operators for object-oriented systems: A survey. In *The 2nd International Conference on Computer and Automation Engineering (ICCAE)* (pp. 614–618). volume 2. 10.1109/ICCAE.2010.5451692.

Ammann, P., Delamaro, M. E., & Offutt, J. (2014). Establishing theoretical minimal sets of mutants. In *Proceedings of the 2014 IEEE International Conference on Software Testing, Verification, and Validation* ICST '14 (pp. 21–30). Washington, DC, USA: IEEE Computer Society. 10.1109/ICST.2014.13.

Banzi, A. S., Nobre, T., Pinheiro, G. B., Árias, J. C. G., Pozo, A., & Vergilio, S. R. (2012). Selecting mutation operators with a multiobjective approach. *Expert Systems with Applications*, *39*, 12131–12142. 10.1016/j.eswa.2012.04.041.

Barbosa, E. F., Maldonado, J. C., & Vincenzi, A. M. R. (2001). Toward the determination of sufficient mutant operators for C. *Software Testing, Verification and Reliability*, *11*, 113–136. 10.1002/stvr.226.

Delgado-Pérez, P., Medina-Bulo, I., Domínguez-Jiménez, J. J., García-Domínguez, A., & Palomo-Lozano, F. (2015). Class mutation operators for C++ object-oriented systems. *Annals of Telecommunications*, *70*, 137–148. 10.1007/s12243-014-0445-4.

Delgado-Pérez, P., Medina-Bulo, I., Palomo-Lozano, F., García-Domínguez, A., & Domínguez-Jiménez, J. J. (2017a). Assessment of class mutation operators for C++ with the MuCPP mutation system. *Information and Software Technology*. *81*, 169–184. 10.1016/j.infsof.2016.07.002.

Delgado-Pérez, P., Segura, S., & Medina-Bulo, I. (2017b). Assessment of C++ object-oriented mutation operators: A selective mutation approach. *Software Testing, Verification and Reliability*, *27*. 10.1002/stvr.1630.

DeMillo, R., Lipton, R., & Sayward, F. (1978). Hints on test data selection: Help for the practicing programmer. *Computer*, *11*, 34–41. 10.1109/C-M.1978.218136.

Derezińska, A. (2003). Object-oriented mutation to assess the quality of tests. In *Proceedings of the 29th Conference on EUROMICRO* (pp. 417–420). Belek, Turkey: IEEE Computer Society.

Derezińska, A. (2006). Quality assessment of mutation operators dedicated for C# programs. In P. Kellenberger (Ed.), *Proceedings of VI International Conference on Quality Software* (pp. 227–234). Beijing (China): IEEE Computer Society. 10.1109/QSIC.2006.51 ISSN 1550-6002.

Derezinska, A., & Halas, K. (2014). Experimental evaluation of mutation testing approaches to Python programs. In *IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* (pp. 156–164). 10.1109/ICSTW.2014.24.

Derezińska, A., & Rudnik, M. (2012). Quality evaluation of object-oriented and standard mutation operators applied to C# programs. In C. Furia, & S. Nanz (Eds.), *Objects, Models, Components, Patterns* (pp. 42–57). Springer Berlin Heidelberg volume 7304 of *Lecture Notes in Computer Science*. 10.1007/978-3-642-30561-0_5.

Derezińska, A., & Szustek, A. (2007). CREAM - A system for object-oriented mutation of C# programs. *Annals Gdansk University of Technology Faculty of ETI*, (pp. 389–406).

Derezińska, A., & Szustek, A. (2012). Object-oriented testing capabilities and performance evaluation of the C# mutation system. In *Proceedings of the 4th IFIP TC 2 Central and East European Conference on Advances in Software Engineering Techniques* CEE-SET'09 (pp. 229–242). Berlin, Heidelberg: Springer-Verlag. 10.1007/978-3-642-28038-2_18.

Dolphin (2017). Dolphin. https://www.kde.org/applications/system/dolphin. [Online; accessed 12-Nov-2017].

Estero-Botaro, A., Palomo-Lozano, F., & Medina-Bulo, I. (2010). Quantitative evaluation of mutation operators for WS-BPEL compositions. In *Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW), 2010* (pp. 142–150). 10.1109/ICSTW.2010.36.

Estero-Botaro, A., Palomo-Lozano, F., Medina-Bulo, I., Domínguez-Jiménez, J. J., & García-Domínguez, A. (2015). Quality metrics for mutation testing with applications to WS-BPEL compositions. *Software Testing, Verification and Reliability*, *25*, 536–571. 10.1002/stvr.1528.

Gopinath, R., Alipour, A., Ahmed, I., Jensen, C., & Groce, A. (2015). How hard does mutation analysis have to be, anyway? In *IEEE 26th International*

Symposium on Software Reliability Engineering ISSRE 2015 (pp. 216–227). 10.1109/ISSRE.2015.7381815.

Hu, J., Li, N., & Offutt, J. (2011). An analysis of OO mutation operators. In *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2011* (pp. 334–341). 10.1109/ICSTW.2011.47.

Inozemtseva, L., & Holmes, R. (2014). Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering* ICSE 2014 (pp. 435–445). New York, NY, USA: ACM. 10.1145/2568225.2568271.

Jia, Y., & Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, *37*, 649–678. 10.1109/TSE.2010.62.

Just, R., & Schweiggert, F. (2015). Higher accuracy and lower run time: Efficient mutation analysis using non-redundant mutation operators. *Software Testing, Verification and Reliability*, *25*, 490–507. 10.1002/stvr.1561.

KMyMoney (2017). KMyMoney, version 4.6.4. https://sourceforge.net/projects/kmymoney2/. [Online; accessed 12-Nov-2017].

Kurtz, B., Ammann, P., Offutt, J., Delamaro, M. E., Kurtz, M., & Gökçe, N. (2016) Analyzing the validity of selective mutation with dominator mutants. Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016 (pp. 571–582). New York, NY, USA: ACM. 10.1145/2950290.2950322.

Ma, Y.-S., Kwon, Y. R., & Kim, S.-W. (2009). Statistical investigation on class mutation operators. *ETRI Journal*, *31*, 140–150. 10.4218/etrij.09.0108.0356.

Ma, Y. S., Kwon, Y. R., & Offutt, A. J. (2002). Inter-class mutation operators for Java. In S. Kawada (Ed.), *Proceedings of XIII International Symposium on Software Reliability Engineering* ISSRE 2002 (pp. 352–363). Annapolis (Maryland): IEEE Computer Society. 10.1109/ISSRE.2002.1173287.

Ma, Y.-S., Offutt, J., & Kwon, Y. R. (2005). MuJava: An automated class mutation system: Research articles. *Software Testing, Verification and Reliability*, *15*, 97–133. 10.1002/stvr.v15:2.

Mateo, P. R., & Usaola, M. P. (2012). Bacterio: Java mutation testing tool: a framework to evaluate quality of tests cases. In *28th IEEE International Conference on Software Maintenance (ICSM), 2012* (pp. 646–649). 10.1109/ICSM.2012.6405344.

Mateo, P. R., & Usaola, M. P. (2015). Reducing mutation costs through uncovered mutants. *Software Testing, Verification and Reliability*, *25*, 464–489. 10.1002/stvr.1534.

Matrix TCL Pro (2017). Matrix TCL Pro, version 2.2. http://www.techsoftpl.com/matrix/download.php. [Online; accessed 12-Nov-2017].

Mresa, E. S., & Bottaci, L. (1999). Efficiency of mutation operators and selective mutation strategies: an empirical study. *Software Testing, Verification and Reliability*, *9*, 205–232. 10.1002/(SICI)1099-1689(199912)9:4<205::AID-STVR186>3.0.CO;2-X.

Namin, A. S., Andrews, J. H., & Murdoch, D. J. (2008). Sufficient mutation operators for measuring test effectiveness. In *ACM/IEEE 30th International Conference on Software Engineering, 2008. ICSE '08* (pp. 351–360). 10.1145/1368088.1368136.

Namin, A. S., Xue, X., Rosas, O., & Sharma, P. (2015). MuRanker: a mutant ranking tool. *Software Testing, Verification and Reliability*, *25(5-7)*, 572–604. 10.1002/stvr.1542.

Offutt, A. J., Lee, A., Rothermel, G., Untch, R. H., & Zapf, C. (1996). An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, *5*, 99–118. 10.1145/227607.227610.

Offutt, A. J., Rothermel, G., & Zapf, C. (1993). An experimental evaluation of selective mutation. In *Proceedings of 15th International Conference on Software Engineering, 1993* (pp. 100–107). 10.1109/ICSE.1993.346062.

Offutt, J. (2011). A mutation carol: past, present and future. *Information and Software Technology*, *53*, 1098–1107. 10.1016/j.infsof.2011.03.007. Special section on mutation testing.

Papadakis, M., Delamaro, M., & Le Traon, Y. (2014). Mitigating the effects of equivalent mutants with mutant classification strategies. *Science of Computer Programming*, *95*, 298–319. 10.1016/j.scico.2014.05.012.

Papadakis, M., & Malevris, N. (2011). Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing. *Software Quality Journal*, *19*, 691–723. 10.1007/s11219-011-9142-y.

QtDOM (2017). QtDOM. https://github.com/qtproject/qtbase/tree/dev/src/xml/dom. [Online; accessed 12-Nov-2017].

Schuler, D., & Zeller, A. (2009). Javalanche: Efficient mutation testing for Java. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* ESEC/FSE '09 (pp. 297–298). New York, NY, USA: ACM. 10.1145/1595696.1595750.

Segura, S., Hierons, R. M., Benavides, D., & Ruiz-Cortés, A. (2011). Mutation testing on an object-oriented framework: An experience report. *Information and Software Technology*, *53*, 1124–1136. 10.1016/j.infsof.2011.03.006. Special Section on Mutation Testing.

Smith, B. H., & Williams, L. (2009). On guiding the augmentation of an automated test suite via mutation analysis. *Empirical Software Engineering*, *14*, 341–369. 10.1007/s10664-008-9083-7.

Tinyxml2 (2017). Tinyxml2. https://github.com/leethomason/tinyxml2. [Online; accessed 12-Nov-2017].

Untch, R. H., Offutt, A. J., & Harrold, M. J. (1993). Mutation analysis using mutant schemata. In *Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis* ISSTA '93 (pp. 139–148). New York, NY, USA: ACM. 10.1145/154183.154265.

Vincenzi, A. M. R., Simão, A. S., Delamaro, M. E., & Maldonado, J. C. (2006). Muta-Pro: Towards the definition of a mutation testing process. *Journal of*

the Brazilian Computer Society, *12*, 49–61. 10.1007/BF03192394.

Wong, W. E., & Mathur, A. P. (1993). *Reducing the Cost of Mutation Testing: An Empirical Study*. techreport Purdue University West Lafayette, Indiana.

XmlRPC (2017). XmlRPC, version 0.7. http://xmlrpcpp.sourceforge.net/. [Online; accessed 12-Nov-2017].

Yao, X., Harman, M., & Jia, Y. (2014). A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *Proceedings of the 36th International Conference on Software Engineering* ICSE 2014 (pp. 919–930). New York, NY, USA: ACM. 10.1145/2568225.2568265.

Zhang, L., Hou, S.-S., Hu, J.-J., Xie, T., & Mei, H. (2010). Is operator-based mutant selection superior to random mutant selection? In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1* ICSE '10 (pp. 435–444). New York, NY, USA: ACM. 10.1145/1806799.1806863.

Zhang, L., Marinov, D., Zhang, L., & Khurshid, S. (2012). Regression mutation testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis* ISSTA 2012 (pp. 331–341). New York, NY, USA: ACM. 10.1145/2338965.2336793.